CEA Comité Español de Automática
XLV JORNADAS AUTOMÁTICA MÁLAGA 2024
UNIVERSIDADE DA CORUÑA

Jornadas de Automática

A user-friendly point cloud processing pipeline for interfacing PCL with YARP

Łukawski, B.*, Rodríguez-Sanz, A., Menendez, E., Victores, J. G., Balaguer, C.

RoboticsLab, Department of Systems Engineering and Automation, Universidad Carlos III de Madrid, Avda. Universidad, 30, 28911, Leganés, Spain.

To cite this article: Łukawski, B., Rodríguez-Sanz, A., Menendez, E., Victores, J. G., Balaguer, C. 2024. A user-friendly point cloud processing pipeline for interfacing PCL with YARP.
Jornadas de Automática, 45. https://doi.org/10.17979/ja-cea.2024.45.10925

Resumen

PCL es una librería de código abierto diseñada para el procesamiento de nubes de puntos. Por otra parte, YARP es un marco de desarrollo e intermediario entre componentes hardware (p. ej. motores y sensores) para elaborar aplicaciones robóticas de alto nivel. Este trabajo presenta una librería de C++ que emplea los algoritmos de PCL sin necesidad de exponer ninguno de sus tipos. Esta permite describir sencillamente y mediante texto secuencias de pasos para el procesamiento de nubes de puntos, exponiendo el resultado final a través de interfaces YARP. Se consigue encapsular así todos los detalles internos de PCL y se evita la dependencia de sus módulos y cabeceras en las aplicaciones cliente. La librería ha sido probada en una aplicación de reconstrucción de escenas para el popular algoritmo KinectFusion, en un módulo de construcción de mallas en tiempo real para el simulador OpenRAVE, y se prevé su uso en tareas de visión con la nueva cabeza del robot humanoide TEO.

Palabras clave: Tecnología robótica, Percepción y sensorización, Información y fusión sensorial, Navegación, programación y visión robótica, Integración sensorial y percepción.

Abstract

PCL is an open-source library and toolkit devoted to point cloud processing. On the other hand, YARP constitutes a robotics middleware and framework which is used to interface with hardware components (such as motors and a wide range of sensors) to build high-level distributed applications. This work presents a C++ library that leverages PCL algorithms without the need of exposing any public PCL types. Sequences of cloud processing steps can be easily described and parsed from text, then executed in a pipeline, and finish with the result being exposed through YARP interfaces, thus encapsulating all PCL-related internals and avoiding the dependency on PCL modules and headers in client applications. The library has been tested on a scene reconstruction app implementing the popular KinectFusion algorithm, on a real-time surface mesh construction module for the OpenRAVE simulator, and it is devised to power vision-oriented tasks on the newly designed head of the humanoid robot TEO.

Keywords: Robotics technology, Perception and sensing, Information and sensor fusion, Robot navigation, programming and vision, Sensor integration and perception.

---

1. Introduction

PCL allows performing a wide range of point cloud processing tasks. However, programming knowledge is required to prepare applications dependent on PCL types in compile-time (due to C++ generics). This may hinder developing flexible and code-less tools, for instance, graphical interfaces that do not know the point type beforehand (i.e. in runtime).

In this work, a new C++ library is described that addresses the aforementioned issues. First, the main software dependencies are briefly introduced. Then, the implementation of the library is explained, and a set of new tools and applications that adopt it is outlined. The humanoid robot these tools aim to enrich is put in context towards the ongoing hardware replacements that will benefit from said library. Finally, some conclusions are drawn and future plans are listed.

*Corresponding author: blukawsk@ing.uc3m.es
Attribution-NonCommercial-ShareAlike 4.0 International (CC BY-NC-SA 4.0)

1.1. Frameworks and Toolkits

PCL (Point Cloud Library) is a community-driven open-source project initially released in 2011 and available under a free license. It comprises a collection of modules and algorithms for 2D/3D point cloud processing. PCL is written in C++ and is designed to be highly modular, with a focus on performance and scalability (Rusu and Cousins, 2011). Its algorithms span a variety of applications, such as (see Table 4): affine transformations, cloud resampling, cloud filtering, cloud processing, mesh construction, surface reconstruction, normal estimation, and mesh simplification.

YARP (Yet Another Robot Platform) is a C++ library intended as both a middleware and framework for robotic applications (Metta et al., 2006). It is designed to provide a communication layer between hardware components (e.g. motors, sensors) to perform low-level control for high-level tasks on distributed environments. A wide range of sensors, such as RGB and depth cameras, can be interfaced through YARP classes and tools by means of its extensive APIs. For the purpose of this work, the `yarp::sig::PointCloud` class will be the main entry point for the point cloud operations described next. YARP is open-source and is actively maintained by the robotics community.

2. Implementation

The proposed "YarpCloudUtils" library (RoboticsLab, 2024b) comprises the following set of C++ free functions:

- `savePLY`: writes a triangular polygon mesh or a point cloud to a file on disk
- `loadPLY`: reads a triangular polygon mesh or a point cloud from a file on disk
- `meshFromCloud`: constructs a triangular polygon mesh from a point cloud
- `processCloud`: transforms an input point cloud into a different output point cloud

By design, all input and output parameters are either standard or YARP types in order to avoid exposing the internal implementation, which depends on a large number of PCL types and modules. Thanks to this, the client code is not required to include any PCL headers nor link against PCL libraries.

The `savePLY` and `loadPLY` helper functions have been implemented using the header-only "tinyply" library, which is a public domain implementation of the PLY mesh format (Diakopoulos, 2020).

The `meshFromCloud` and `processCloud` functions implement a pipeline-based approach to cloud processing. Both accept an input point cloud; the former produces a mesh, while the latter returns a different point cloud, adhering to the sequence of steps encoded in a configuration parameter also passed to these functions. The pipeline can be textually described in a configuration file, which is parsed by YARP utilities. Listing 1 describes a sample pipeline which consists of three steps: downsampling using the PCL "VoxelGrid" algorithm (which maps to a C++ class of the same name), normal estimation, and the final surface reconstruction.

Listing 1: Sample pipeline configuration file.

```
[myPipeline downsample]
algorithm "VoxelGrid"
leafSize 0.02f

[myPipeline estimate]
algorithm "NormalEstimationOMP"
kSearch 40

[myPipeline reconstruct]
algorithm "Poisson"
```

By further exploiting YARP capabilities of configuration parsing, the same pipeline can be described using the command-line interface as in Listing 2. Here, the sample application "app" has been prepared to expect an arbitrary number of parameters which are then parsed to define the cloud processing pipeline.

Listing 2: Pipeline configuration via command-line interface.

```
app --myPipeline downsample estimate reconstruct \
    --downsample::algorithm VoxelGrid \
    --downsample::leafSize 0.02f \
    --estimate::algorithm NormalEstimationOMP \
    --estimate::kSearch 40 \
    --reconstruct::algorithm Poisson
```

The internals of the "YarpCloudUtils" library heavily rely on C++ generic programming, that is, the usage of templates in the library's implementation is widespread and therefore several design decisions had to be taken in order to prepare for dealing with the compile-time intricacies of this paradigm. For instance, an equivalence had to be established between the YARP types exposed in the public interface of the four main functions, and the PCL types used internally. Both frameworks define their own class template, `yarp::sig::PointCloud<T>` and `pcl::PointCloud<T>` respectively, whose only type parameter refers to the data type of the points stored internally. Table 1 lists the supported equivalences between YARP and PCL point types.

Table 1: YARP and PCL point type equivalences.

YARP type (`yarp::sig::`)	PCL type (`pcl::`)
DataXY*	PointXY*
DataXYZ	PointXYZ
DataNormal*	Normal*
DataXYZRGBA	PointXYZRGB
DataXYZI	PointXYZI
DataInterestPointXYZ	InterestPoint
DataXYZNormal	PointNormal
DataXYZNormalRGBA	PointXYZRGBNormal

Despite being included in the previous table, it must be noted that XY and plain-normal types are not available for surface meshing and cloud processing.

Another consequence of using generics is the inability to resolve the type of the point cloud at runtime through polymorphism. Compile-time switches, known as "traits", had to be introduced to handle the type conversions between YARP and PCL types, as well as between the expected input and output data types of any PCL class invoked within the pipeline.

Since client applications are free to define any custom pipeline through a text descriptor file, the library must both forbid impossible type conversions, and allow the expected ones to happen at any intermediate step of the process. All combinations are considered by the compiler and already built into the library, therefore client applications will perceive no compile-time errors in case an impossible scenario is encountered. Whenever such an incompatible transition between steps is requested, a runtime error will be thrown instead.

To accomodate for the multiple supported PCL point types previously listed, the cloud_container class is introduced to store any of these types, but at most one of them can be used simultaneously (Figure 1).

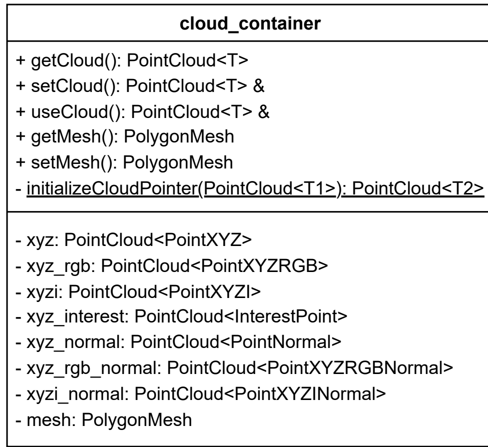


Figure 1: UML diagram for the *cloud_container* class.

The private attributes of this class are shared pointers to PointCloud<T> instances of the specific point type T; extra room is also left for a polygon mesh. Public accessors are provided to set at most one of these attributes at any time, depending on the algorithm requested at the current pipeline step, or to perform the conversion from the available type to the requested one.

Algorithm 1 describes the implementation of the function processCloud(), which is responsible for the cloud processing pipeline. It takes an input cloud, an output cloud, and a sequence of steps to be executed. The function first checks if the input and output types are supported, then converts the input cloud to a PCL type (per Table 1), processes it, and finally converts the output cloud back to a YARP type. The meshFromCloud() function follows an analogous algorithm, but it returns a mesh instead of a cloud.

Algorithm 1: Implementation of *processCloud()*.

```
Function processCloud(in, out, steps):
    T_in ← yarp_type_to_pcl_type(in)
    T_out ← yarp_type_to_pcl_type(out)
    if not is_supported<T_in, T_out>() then
        return false
    end
    pcl_in<T_in> ← yarp_to_pcl(in)
    pcl_out<T_out> ← processPCL(pcl_in, steps)
    out ← pcl_to_yarp(pcl_out)
    return true
```

The implementation of the method getCloud<T>() of class cloud_container is outlined in Algorithm 2.

Algorithm 2: Implementation of *getCloud<T>()*.

```
Function getCloud<T>(in):
    if not xyz is empty then
        // initializeCloudPointer()
        if is_same_type(xyz, T) then
            return xyz
        else if not is_convertible(xyz, T) then
            throw runtime_error
        else
            out ← copyPointCloud(xyz)
            return out
        end
    else if not xyz_rgb is empty then
        // repeat previous block
    else if ... then
        // xyzi, xyz_interest...
    else
        throw runtime_error
    end
```

It is assumed that one of the private attributes was previously initialized through setCloud<T>(). If the requested type T matches the available one when getCloud<T>() is called, the attribute is directly returned by the function. Otherwise, a fallback conversion is attempted, and if it fails, a runtime error is thrown. The supported fallbacks are listed in Table 2. Compound types can be stripped from the excess information they carry, e.g. a XYZ+RGB+normal type can fall back to either XYZ+RGB or XYZ+normal. The default type is plain XYZ since all supported point types imply it.

Table 2: Allowed PCL point type fallbacks.

source type	target type
any	pcl::PointXYZ
pcl::PointXYZRGBNormal	pcl::PointXYZRGB
	pcl::PointNormal
pcl::PointXYZINormal	pcl::PointXYZI
	pcl::PointNormal

Algorithm 3 describes the implementation of the internal processPCL() function, which is responsible for the actual cloud processing. It iterates over the requested sequence of steps, each of which invokes the corresponding PCL class that implements the desired algorithm. At this stage, all involved data types are PCL types that have undergone the preliminary conversions and fallbacks.

Algorithm 3: Implementation of *processPCL()*.

```
Function processPCL(in, out, steps):
    obj ← cloud_container()
    obj.setCloud(in)
    for step in steps do
        obj ← processStep(obj, step)
    end
    out ← obj.getCloud()
```

Table 3: PCL point type decay conversions.

category	source type	target type
any	any	same as source
XYZ(+RGB)	`pcl::PointXYZRGB` `pcl::PointXYZRGBNormal`	`pcl::PointXYZRGB`
	other	`pcl::PointXYZ`
XYZ+RGB	any	`pcl::PointXYZRGB`

XYZI(+normal)	`pcl::PointNormal`
	`pcl::PointXYZRGBNormal`
	`pcl::PointXYZINormal`
		`pcl::PointXYZINormal`

	other	`pcl::PointXYZI`
XYZ+normal	`pcl::PointXYZ` `pcl::InterestPoint`	`pcl::PointNormal`
	`pcl::PointXYZRGB`	`pcl::PointXYZRGBNormal`
	`pcl::PointXYZI`	`pcl::PointXYZINormal`
	other	same as source

In addition, a decay conversion is performed to ensure that the output type of each step matches the input type of the next one, and to prepare `cloud_container` for storing the resulting output type of the latter. Table 3 lists the supported decay conversions. The "XYZ(+RGB)" notation stands for XYZ or XYZ+RGB point types, whereas "XYZ+RGB" denotes that the desired point type must carry information for both XYZ and RGB channels.

For instance, the "VoxelGrid" algorithm doesn't enforce a specific type requirement on its input, hence any point type queried from the current `cloud_container` instance via `getCloud()` and passed to it will be accepted. Besides, this same type will be the expected output type passed to the `cloud_container` instance of the following step via `setCloud()`. The "NormalEstimation" algorithm, on the other hand, produces a point cloud that aditionally stores the normals, therefore the corresponding attribute of the `cloud_container` instance will be initialized accordingly (`PointXYZRGBNormal` if the input was `PointXYZRGB`, `PointXYZINormal` if the input was `PointXYZI`, and so on). The "Poisson" algorithm expects a cloud with normal points, whatever the base type would be (e.g `PointNormal`, `PointXYZRGBNormal`, or `PointXYZINormal`), and its result is a `PolygonMesh`.

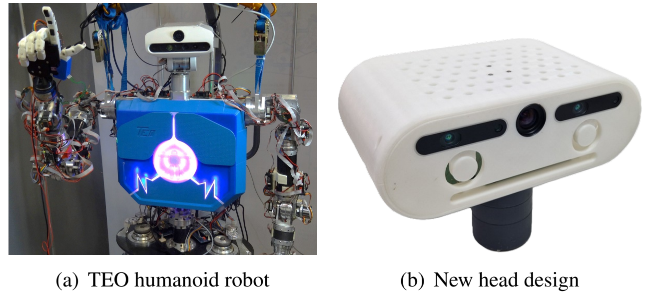
Table 4 lists the supported PCL algorithms, their expected input types, and the output types they produce, according to the categories defined in Table 3.

3. Case Studies

The tools described in this work are intended for use on the TEO platform developed by the RoboticsLab group of Universidad Carlos III de Madrid (Martínez et al., 2012). This full-sized humanoid robot, depicted in Figure 2(a), features 28 degrees of freedom distributed along its four limbs, neck and torso. A range of sensors allow it to interface with its environment, including a high-resolution RGB camera Flea3 FL3-U3-88S2C-C by Point Grey, and an ASUS XtionPRO Live RGB-depth sensor, both mounted on its articulated head. YARP powers low-level and high-level tasks interfacing with these hardware components over a distributed local network.

3.1. *Redesign of TEO's Head*

A prototype for a new head has been developed which integrates an improved sensing system with two Intel Realsense D435if RGBD sensors, featuring IR pass filters for enhanced depth noise quality and avoidance of false object detection due to light reflections, as well as an inertial measurement unit (IMU) for better image stabilization, for instance. The high-resolution Flea3 camera remains in the new design to aid in focusing on narrow fields of view for static mapping (see Figure 2(b)). The new head also features an advanced computing module Nvidia Jetson AGX Xavier for developing vision and AI applications, which can be used for energy-efficient point cloud generation of the two-camera setup.



(a) TEO humanoid robot (b) New head design

Figure 2: Humanoid platform designed by RoboticsLab-UC3M.

The new head poses an opportunity for the development of novel vision-related applications which use all three cameras. An improvement to the construction of 3D maps as proposed in this work might come from the addition of color information to the point cloud. This is shown to improve the performance of the algorithms where groups of points are similar based on the geometric features (Han et al., 2024).

A paradigm shift comes from the inclusion of two RGBD cameras at eye-width distance, where the point clouds obtained from each camera need to be synchronized and mixed in those areas where their fields of view collide. Several approaches exist to ensure proper synchronization across multiple RealSense cameras (Yoon et al., 2021) and their accurate arrangement, configuration and data processing in multi-camera setups (Herguedas et al., 2020).

Table 4: Supported PCL algorithms.

PCL class (`pcl::`)	usage	expected type
`transformPointCloud`	affine transformation	any
`transformPointCloudWithNormals`	affine transformation	any normal type
`ApproximateVoxelGrid`	cloud resampling	any
`BilateralFilter`	cloud filtering	XYZI(+normal)
`BilateralUpsampling`	cloud processing	XYZRGBA
`ConcaveHull`	mesh construction	any
`ConvexHull`	mesh construction	any
`CropBox`	cloud filtering	any
`FastBilateralFilter`	cloud filtering	XYZ(+RGBA)
`FastBilateralFilterOMP`	cloud filtering	XYZ(+RGBA)
`GreedyProjectionTriangulation`	mesh construction	XYZ/XYZI/XYZRGBA + normal
`GridMinimum`	cloud resampling	any
`GridProjection`	surface reconstruction	XYZ/XYZI/XYZRGBA + normal
`LocalMaximum`	cloud resampling	any
`MarchingCubesHoppe`	surface reconstruction	XYZ/XYZI/XYZRGBA + normal
`MarchingCubesRBF`	surface reconstruction	XYZ/XYZI/XYZRGBA + normal
`MedianFilter`	cloud filtering	any
`MeshQuadricDecimationVTK`	mesh processing	mesh
`MeshSmoothingLaplacianVTK`	mesh processing	mesh
`MeshSmoothingWindowedSincVTK`	mesh processing	mesh
`MeshSubdivisionVTK`	mesh processing	mesh
`MovingLeastSquares`	cloud processing	any
`NormalEstimation`	normal estimation	any
`NormalEstimationOMP`	normal estimation	any
`OrganizedFastMesh`	mesh construction	XYZ(+RGBA)
`PassThrough`	cloud filtering	any
`Poisson`	surface reconstruction	XYZ/XYZI/XYZRGBA + normal
`RadiusOutlierRemoval`	cloud filtering	any
`RandomSample`	cloud resampling	any
`SamplingSurfaceNormal`	cloud resampling	XYZ/XYZI/XYZRGBA + normal
`ShadowPoints`	cloud filtering	any normal type
`SimplificationRemoveUnusedVertices`	mesh simplification	mesh
`StatisticalOutlierRemoval`	cloud filtering	any
`UniformSampling`	cloud resampling	any
`VoxelGrid`	cloud resampling	any

3.2. Simulation: Real-Time Mesh Visualization

Simulators are extensively used prior to performing any task on the actual robot in real scenarios. A collection of TEO models is available for the OpenRAVE simulator. Its capabilities can be extended through plugins to add new controllers, visualization tools, physics engines, etc. A new YARP device has been implemented to query frames from a depth sensor, and to construct a surface mesh in real time using this information. It connects to the simulated environment through an existing OpenRAVE plugin which was extended to permit mesh visualization. The projection of the mesh is attached to the articulated head, thus following its motion. This application is intended to replicate the joint configuration of the real robot on the simulated one, along with the depth information collected by the camera on the robot's head (RoboticsLab, 2024a).

The "OrganizedFastMesh" PCL algorithm bundled with "YarpCloudUtils" is recommended in this scenario due to performance concerns. It is optimized for point clouds captured from a camera frame. The "SimplificationRemoveUnusedVertices" algorithm helps reducing the size of the resulting mesh.

Figure 3 depicts a surface mesh construction experiment using real-time depth information acquired through an Intel RealSense D435i RGBD camera.

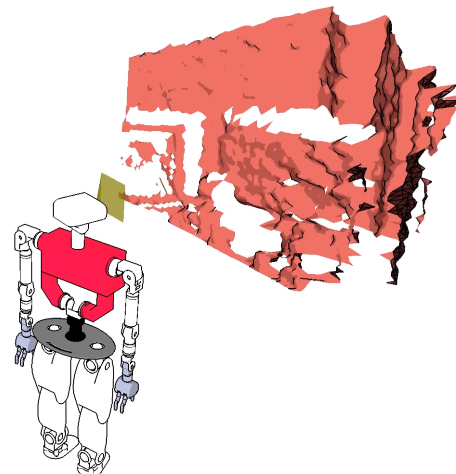


Figure 3: OpenRAVE real-time mesh visualization (Łukawski, 2020).

3.3. High-Level Applications

In addition, a scene reconstruction application was developed implementing the popular KinectFusion algorithm through OpenCV classes (Newcombe et al., 2011; RoboticsLab, 2024b). It is aimed to perform real-time surface mapping and tracking with a depth sensor. By leveraging the capabilities of YARP, a communications layer via a server-client scheme was introduced in order to remotely start and stop the processing, and to query the point cloud on demand from the server application. The "YarpCloudUtils" library presented in this work is used to process said point cloud, to generate a surface mesh from it (Figure 4), and to export it to a PLY file.

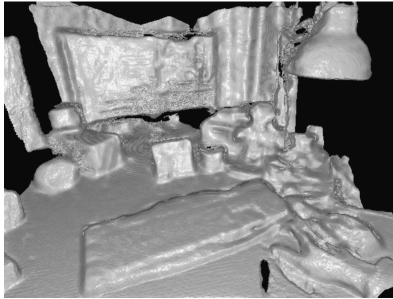
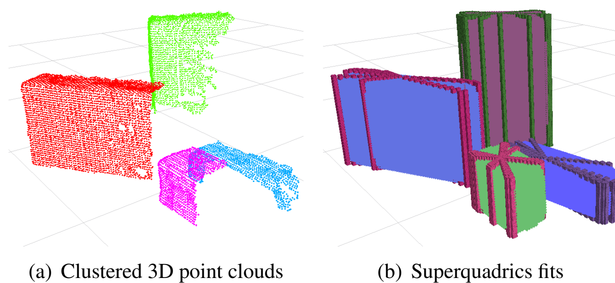


Figure 4: Reconstructed scene using the KinectFusion algorithm.

Lastly, a state-of-the-art object identification technique has been devised to be complemented by the PCL-driven pipeline described in this work. Figure 5 depicts a collection of objects whose point cloud has been conveniently clustered, and their volume delimited with superquadrics fitting.



(a) Clustered 3D point clouds (b) Superquadrics fits

Figure 5: Superquadric estimation with point clouds (Menendez et al., 2024).

4. Conclusions

This work described the implementation of a new library, "YarpCloudUtils", aimed to simplify the usage of PCL algorithms in C++ code. Its public interface was designed to leverage the YARP robotics framework, thus improving the integration of point cloud-related applications with other components in a robotic task. The library was designed to take advantage of YARP capabilities in configuration parameter parsing: a pipeline of cloud processing steps can be stored and read from a text file or via command line options. The main advantage of this strategy is that users don't need to configure the correct PCL modules nor include specific headers. This approach was materialized in a number of case studies involving the RGBD sensor of the TEO humanoid robot, and a new head design was described, focusing on future potential uses of this library.

On a future note, a rotating platform is being developed to allow scanning objects with a fixed RGBD sensor. The resulting point cloud would undergo a processing step with a pipeline configured in the library presented in this work. An alternative setup is also devised, in which a collaborative ABB GoFa robot has such a sensor mounted on its end effector, and performs a scan of a fixed object following a predefined configurable trajectory.

Future plans also include exploring the integration of the current vision-oriented YARP ecosystem with the ROS2 programming stack, the development of an analogous mesh visualization plugin for the Gazebo simulator, and the adoption of C++20 to exploit its new features towards generic programming and "concepts".

Acknowledgments

This research has been financed by ROBOASSET, "Sistemas robóticos inteligentes de diagnóstico y rehabilitación de terapias de miembro superior", PID2020-113508RB-I00, financed by AEI/10.13039/501100011033; "RoboCity2030-DIH-CM, Madrid Robotics DIH", S2018/NMT-4331, financed by "Programas de Actividades I+D en la Comunidad de Madrid"; "iREHAB: AI-powered Robotic Personalized Rehabilitation", ISCIII-AES-2022/003041 financed by ISCIII and EU; and EU structural funds.

References

Diakopoulos, D., 2020. tinyply: C++11 ply 3d mesh format importer & exporter. https://github.com/ddiakopoulos/tinyply.

Han, T., Zhang, R., Kan, J., Dong, R., Zhao, X., Yao, S., 2024. A point cloud registration framework with color information integration. Remote Sensing 16 (5).
DOI: 10.3390/rs16050743

Herguedas, R., López-Nicolás, G., Sagüés, C., 2020. Experimental multi-camera setup for perception of dynamic objects. In: Workshop ROMADO, IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS). pp. 11874–11878.

Łukawski, B., 2020. OpenRAVE: surface mesh from real depth sensor & TEO. https://youtu.be/CqOWiSKAAXo.

Martínez, S., Monje, C. A., Jardón, A., Pierro, P., Balaguer, C., Muñoz, D., 2012. TEO: Full-size humanoid robot design powered by a fuel cell system. Cybernetics and Systems 43 (3), 163–180.
DOI: 10.1080/01969722.2012.659977

Menendez, E., Martínez, S., Díaz-de María, F., Balaguer, C., 2024. Integrating egocentric and robotic vision for object identification using siamese networks and superquadric estimations in partial occlusion scenarios. Biomimetics 9 (2).
DOI: 10.3390/biomimetics9020100

Metta, G., Fitzpatrick, P., Natale, L., 2006. YARP: yet another robot platform. International Journal of Advanced Robotic Systems 3 (1), 43–48.
DOI: 10.5772/5761

Newcombe, R. A., Izadi, S., Hilliges, O., Molyneaux, D., Kim, D., Davison, A. J., Kohi, P., Shotton, J., Hodges, S., Fitzgibbon, A., 2011. KinectFusion: Real-time dense surface mapping and tracking. In: 10th IEEE International Symposium on Mixed and Augmented Reality. pp. 127–136.
DOI: 10.1109/ISMAR.2011.6092378

RoboticsLab, 2024a. OpenRAVE plugins to interface OpenRAVE with YARP. https://github.com/roboticslab-uc3m/openrave-yarp-plugins.

RoboticsLab, 2024b. Vision processing. https://github.com/roboticslab-uc3m/vision.

Rusu, R. B., Cousins, S., 2011. 3D is here: Point Cloud Library (PCL). In: IEEE International Conference on Robotics and Automation (ICRA). IEEE, Shanghai, China, pp. 1–4.
DOI: 10.1109/ICRA.2011.5980567

Yoon, H., Jang, M., Huh, J., Kang, J., Lee, S., 2021. Multiple sensor synchronization with theRealSense RGB-D camera. Sensors 21 (18).
DOI: 10.3390/s21186276